
Authors
ributors
nse
bjects
ct Usage and Properties
Prototype
wnProperty
for in Loop
unctions
ther
}
Foo.prototype = {
method: function(
};
function Bar() {}
// Set Bar's prototy
Bar.prototype = new
Bar.prototype.foo =
// Make sure to lis
Bar.prototype.const
JAVASCRIPT GUIDE

目錄

Google JavaScript 风格指南

转载一下Google JavaScript Style Guide,文章不但指出每条规范，还解释了为什么这样写的原因，同时给出了对与错的实例，写得非常的详细，很值参考，正在考虑应该不应该翻译一下，然后借鉴到项目团队中去，站在巨人的肩膀上会看得更远。

修订版: 2.9

```
Aaron Whyte
Bob Jervis
Dan Pupius
Eric Arvidsson
Fritz Schneider
Robby Walker
```

背景

JavaScript 是一种客户端脚本语言, Google 的许多开源工程中都有用到它. 这份指南列出了编写 JavaScript 时需要遵守的规则.

JavaScript 语言规范

变量

声明变量必须加上 `var` 关键字.

Decision: 当你没有写 `var`, 变量就会暴露在全局上下文中, 这样很可能会和现有变量冲突. 另外, 如果没有加上, 很难明确该变量的作用域是什么, 变量也很可能像在局部作用域中, 很轻易地泄漏到 Document 或者 Window 中, 所以务必用 `var` 去声明变量.

常量

常量的形式如: `NAMES_LIKE_THIS`, 即使用大写字符, 并用下划线分隔. 你也可用 `@const` 标记来指明它是一个常量. 但请永远不要使用 `const` 关键词.

Decision:

对于基本类型的常量, 只需转换命名.

```
/**
 * The number of seconds in a minute.
 * @type {number}
 */
goog.example.SECONDS_IN_A_MINUTE = 60;
```

对于非基本类型, 使用 `@const` 标记.

```
/**
 * The number of seconds in each of the given units.
 * @type {Object.<number>}
 * @const
 */
goog.example.SECONDS_TABLE = {
  minute: 60,
  hour: 60 * 60,
  day: 60 * 60 * 24
}
```

这标记告诉编译器它是常量.

至于关键词 `const`, 因为 IE 不能识别, 所以不要使用.

分号

总是使用分号.

如果仅依靠语句间的隐式分隔, 有时会很麻烦. 你自己更能清楚哪里是语句的起止.

而且有些情况下, 漏掉分号会很危险:

```
// 1.
MyClass.prototype.myMethod = function() {
  return 42;
}  // No semicolon here.

(function() {
  // Some initialization code wrapped in a function to create a sco
})();

var x = {
  'i': 1,
  'j': 2
}  // No semicolon here.

// 2\.  Trying to do one thing on Internet Explorer and another on
// I know you'd never write code like this, but throw me a bone.
[normalVersion, ffVersion][isIE]();

var THINGS_TO_EAT = [apples, oysters, sprayOnCheese]  // No semicol

// 3\. conditional execution a la bash
-1 == resultOfOperation() || die();
```

这段代码会发生些什么诡异事呢?

1. 报 JavaScript 错误 – 例子1上的语句会解释成, 一个函数带一匿名函数作为参数而被调用, 返回42后, 又一次被"调用", 这就导致了错误.
2. 例子2中, 你很可能会在运行时遇到 'no such property in undefined' 错误, 原因是代码试图这样 `x[ffVersion][isIE]()` 执行.
3. 当 `resultOfOperation()` 返回非 NaN 时, 就会调用 `die` , 其结果也会赋给 `THINGS_TO_EAT` .

为什么?

JavaScript 的语句以分号作为结束符, 除非可以非常准确推断某结束位置才会省略分号. 上面的几个例子产出错误, 均是在语句中声明了函数/对象/数组直接量, 但 闭括号('}'或']')并不足以表示该语句的结束. 在 JavaScript 中, 只有当语句后的下一个符号是后缀或括号运算符时, 才会认为该语句的结束.

遗漏分号有时会出现很奇怪的结果, 所以确保语句以分号结束.

嵌套函数

可以使用

嵌套函数很有用, 比如,减少重复代码, 隐藏帮助函数, 等. 没什么其他需要注意的地方, 随意使用.

块内函数声明

不要在块内声明一个函数

不要写成:

```
if (x) {
  function foo() {}
}
```

虽然很多 JS 引擎都支持块内声明函数, 但它不属于 ECMAScript 规范 (见 ECMA-262, 第13和14条). 各个浏览器糟糕的实现相互不兼容, 有些也与未来 ECMAScript 草案相违背. ECMAScript 只允许在脚本的根语句或函数中声明函数. 如果确实需要在块中定义函数, 建议使用函数表达式来初始化变量:

```
if (x) {
  var foo = function() {}
}
```

异常

可以

你在写一个比较复杂的应用时, 不可能完全避免不会发生任何异常. 大胆去用吧.

自定义异常

可以

有时发生异常了, 但返回的错误信息比较奇怪, 也不易读. 虽然可以将含错误信息的引用对象或者可能产生错误的完整对象传递过来, 但这样做都不是很好, 最好还是自定义异常类, 其实这些基本上都是最原始的异常处理技巧. 所以在适当的时候使用自定义异常.

标准特性

总是优于非标准特性.

最大化可移植性和兼容性, 尽量使用标准方法而不是用非标准方法, (比如, 优先用 `string.charAt(3)` 而不用 `string[3]` , 通过 DOM 原生函数访问元素, 而不是使用应用封装好的快速接口.

封装基本类型

不要

没有任何理由去封装基本类型, 另外还存在一些风险:

```
var x = new Boolean(false);
if (x) {
  alert('hi');  // Shows 'hi'.
}
```

除非明确用于类型转换, 其他情况请千万不要这样做！

```
var x = Boolean(0);
if (x) {
  alert('hi');  // This will never be alerted.
}
typeof Boolean(0) == 'boolean';
typeof new Boolean(0) == 'object';
```

有时用作 `number` , `string` 或 `boolean` 时, 类型的转换会非常实用.

多级原型结构

不是首选

多级原型结构是指 JavaScript 中的继承关系. 当你自定义一个D类, 且把B类作为其原型, 那么这就获得了一个多级原型结构. 这些原型结构会变得越来越复杂!

使用the Closure 库中的 `goog.inherits()` 或其他类似的用于继承的函数, 会是更好的选择.

```
function D() {
  goog.base(this)
}
goog.inherits(D, B);

D.prototype.method = function() {
  ...
};
```

方法定义

`Foo.prototype.bar = function() { ... };`

有很多方法可以给构造器添加方法或成员，我们更倾向于使用如下的形式:

```
Foo.prototype.bar = function() {
  /* ... */
};
```

闭包

可以, 但小心使用.

闭包也许是 JS 中最有用的特性了.

有一份比较好的介绍闭包原理的文档.

有一点需要牢记, 闭包保留了一个指向它封闭作用域的指针, 所以, 在给 DOM 元素附加闭包时, 很可能会产生循环引用, 进一步导致内存泄漏. 比如下面的代码:

```
function foo(element, a, b) {
  element.onclick = function() { /* uses a and b */ };
}
```

这里, 即使没有使用 `element`, 闭包也保留了 `element`, `a` 和 `b` 的引用, . 由于 `element` 也保留了对闭包的引用, 这就产生了循环引用, 这就不能被 GC 回收.

这种情况下, 可将代码重构为:

```
function foo(element, a, b) {
  element.onclick = bar(a, b);
}

function bar(a, b) {
  return function() { /* uses a and b */ }
}
```

eval()

只用于解析序列化串 (如: 解析 RPC 响应)

`eval()` 会让程序执行的比较混乱, 当 `eval()` 里面包含用户输入的话就更加危险.

可以用其他更佳的, 更清晰, 更安全的方式写你的代码, 所以一般情况下请不要使用 eval().

当碰到一些需要解析序列化串的情况下(如, 计算 RPC 响应), 使用 `eval` 很容易实现.

解析序列化串是指将字节流转换成内存中的数据结构. 比如, 你可能会将一个对象输出成文件形式:

```
users = [
  {
    name: 'Eric',
    id: 37824,
    email: 'jellyvore@myway.com'
  },
  {
    name: 'xtof',
    id: 31337,
    email: 'b4d455h4x0r@google.com'
  },
  ...
];
```

很简单地调用 `eval` 后, 把表示成文件的数据读取回内存中.

类似的, `eval()` 对 RPC 响应值进行解码. 例如, 你在使用 `XMLHttpRequest` 发出一个 RPC 请求后, 通过 eval () 将服务端的响应文本转成 JavaScript 对象:

```
var userOnline = false;
var user = 'nusrat';
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET', 'http://chat.google.com/isUserOnline?user=' + u
xmlhttp.send('');
// Server returns:
// userOnline = true;
if (xmlhttp.status == 200) {
  eval(xmlhttp.responseText);
}
// userOnline is now true.
```

`with() {}`

不要使用

使用 `with` 让你的代码在语义上变得不清晰. 因为 `with` 的对象, 可能会与局部变量产生冲突, 从而改变你程序原本的用义.

下面的代码是干嘛的?

```
with (foo) {
  var x = 3;
  return x;
}
```

答案: 任何事. 局部变量 `x` 可能被 `foo` 的属性覆盖, 当它定义一个 setter 时, 在赋值 `3` 后会执行很多其他代码.

所以不要使用 `with` 语句.

this

仅在对象构造器, 方法, 闭包中使用.

`this` 的语义很特别. 有时它引用一个全局对象(大多数情况下), 调用者的作用域(使用 `eval` 时), DOM 树中的节点(添加事件处理函数时), 新创建的对象(使用一个构造器), 或者其他对象(如果函数被 `call()` 或 `apply()`).

使用时很容易出错, 所以只有在下面两个情况时才能使用:

- 在构造器中
- 对象的方法(包括创建的闭包)中

for-in 循环

只用于 object/map/hash 的遍历

对 `Array` 用 `for-in` 循环有时会出错. 因为它并不是从 `0` 到 `length - 1` 进行遍历, 而是所有出现在对象及其原型链的键值. 下面就是一些失败的使用案例:

```
function printArray(arr) {
  for (var key in arr) {
    print(arr[key]);
  }
}

printArray([0,1,2,3]);  // This works.

var a = new Array(10);
printArray(a);  // This is wrong.

a = document.getElementsByTagName('*');
printArray(a);  // This is wrong.

a = [0,1,2,3];
a.buhu = 'wine';
printArray(a);  // This is wrong again.

a = new Array;
a[3] = 3;
printArray(a);  // This is wrong again.
```

而遍历数组通常用最普通的 for 循环.

```
function printArray(arr) {
  var l = arr.length;
  for (var i = 0; i < l; i++) {
    print(arr[i]);
  }
}
```

关联数组

永远不要使用 `Array` 作为 map/hash/associative 数组.

数组中不允许使用非整型作为索引值, 所以也就不允许用关联数组. 而取代它使用 `Object` 来表示 map/hash 对象.

`Array` 仅仅是扩展自 `Object` (类似于其他 JS 中的对象, 就像 `Date` , `RegExp` 和 `String`)一样来使用.

多行字符串

不要使用

不要这样写长字符串:

```
var myString = 'A rather long string of English text, an error mess
                actually that just keeps going and going -- an erro
                message to make the Energizer bunny blush (right th
                those Schwarzenegger shades)! Where was I? Oh yes,
                you\'ve got an error and all the extraneous whitesp
                just gravy.  Have a nice day.';
```

在编译时, 不能忽略行起始位置的空白字符; "\" 后的空白字符会产生奇怪的错误; 虽然大多数脚本引擎支持这种写法, 但它不是 ECMAScript 的标准规范.

Array 和 Object 直接量

使用

使用 `Array` 和 `Object` 语法, 而不使用 `Array` 和 `Object` 构造器.

使用 Array 构造器很容易因为传参不恰当导致错误.

```
// Length is 3.
var a1 = new Array(x1, x2, x3);

// Length is 2.
var a2 = new Array(x1, x2);

// If x1 is a number and it is a natural number the length will be
// If x1 is a number but not a natural number this will throw an ex
// Otherwise the array will have one element with x1 as its value.
var a3 = new Array(x1);

// Length is 0.
var a4 = new Array();
```

如果传入一个参数而不是2个参数, 数组的长度很有可能就不是你期望的数值了.

为了避免这些歧义，我们应该使用更易读的直接量来声明.

```
var a = [x1, x2, x3];
var a2 = [x1, x2];
var a3 = [x1];
var a4 = [];
```

虽然 Object 构造器没有上述类似的问题, 但鉴于可读性和一致性考虑, 最好还是在字面上更清晰地指明.

```
var o = new Object();

var o2 = new Object();
o2.a = 0;
o2.b = 1;
o2.c = 2;
o2['strange key'] = 3;
```

应该写成:

```
var o = {};

var o2 = {
  a: 0,
  b: 1,
  c: 2,
  'strange key': 3
};
```

修改内置对象的原型

不要

千万不要修改内置对象，如 `Object.prototype` 和 `Array.prototype` 的原型. 而修改内置对象，如 `Function.prototype` 的原型，虽然少危险些，但仍会导致调试时的诡异现象.

所以也要避免修改其原型.

IE下的条件注释

不要使用

不要这样子写:

```
var f = function () {
    /*@cc_on if (@_jscript) { return 2* @*/  3; /*@ } @*/
};
```

条件注释妨碍自动化工具的执行, 因为在运行时, 它们会改变 JavaScript 语法树.

JavaScript 编码风格

命名

通常，使用 `functionNamesLikeThis`，`variableNamesLikeThis`，`ClassNamesLikeThis`，`EnumNamesLikeThis`，`methodNamesLikeThis`，和 `SYMBOLIC_CONSTANTS_LIKE_THIS`.

展开见细节.

属性和方法

- 文件或类中的 私有 属性，变量和方法名应该以下划线 “_” 开头.
- 保护 属性，变量和方法名不需要下划线开头，和公共变量名一样.

更多有关 私有 和 保护的信息见，visibility.

方法和函数参数

可选参数以 `opt_` 开头.

函数的参数个数不固定时，应该添加最后一个参数 `var_args` 为参数的个数. 你也可以不设置 `var_args` 而取代使用 `arguments`.

可选和可变参数应该在 `@param` 标记中说明清楚. 虽然这两个规定对编译器没有任何影响，但还是请尽量遵守

Getters 和 Setters

Getters 和 setters 并不是必要的. 但只要使用它们了，就请将 getters 命名成 `getFoo()` 形式，将 setters 命名成 `setFoo(value)` 形式. (对于布尔类型的 getters，使用 `isFoo()` 也可.)

命名空间

JavaScript 不支持包和命名空间.

不容易发现和调试全局命名的冲突，多个系统集成时还可能因为命名冲突导致很严重的问题.

为了提高 JavaScript 代码复用率，我们遵循下面的约定以避免冲突.

为全局代码使用命名空间

在全局作用域上，使用一个唯一的，与工程/库相关的名字作为前缀标识. 比如，你的工程是 “Project Sloth”，那么命名空间前缀可取为 `sloth.*`.

```
var sloth = {};

sloth.sleep = function() {
  ...
};
```

许多 JavaScript 库, 包括the Closure LibraryandDojo toolkit为你提供了声明你自己的命名空间的函数. 比如:

```
goog.provide('sloth');

sloth.sleep = function() {
  ...
};
```

明确命名空间所有权

当选择了一个子命名空间, 请确保父命名空间的负责人知道你在用哪个子命名空间, 比如说, 你为工程 'sloths' 创建一个 'hats' 子命名空间, 那确保 Sloth 团队人员知道你在使用 `sloth.hats` .

外部代码和内部代码使用不同的命名空间

"外部代码" 是指来自于你代码体系的外部, 可以独立编译. 内外部命名应该严格保持独立.

如果你使用了外部库, 他的所有对象都在 `foo.hats.*` 下, 那么你自己的代码不能在 `foo.hats.*` 下命名, 因为很有可能其他团队也在其中命名.

```
foo.require('foo.hats');

/**
 * WRONG -- Do NOT do this.
 * @constructor
 * @extend {foo.hats.RoundHat}
 */
foo.hats.BowlerHat = function() {
};
```

如果你需要在外部命名空间中定义新的 API, 那么你应该直接导出一份外部库, 然后在这份代码中修改.

在你的内部代码中, 应该通过他们的内部名字来调用内部 API , 这样保持一致性可让编译器更好的优化你的代码.

```
foo.provide('googleyhats.BowlerHat');

foo.require('foo.hats');

/**
 * @constructor
 * @extend {foo.hats.RoundHat}
 */
googleyhats.BowlerHat = function() {
  ...
};

goog.exportSymbol('foo.hats.BowlerHat', googleyhats.BowlerHat);
```

重命名那些名字很长的变量, 提高可读性

主要是为了提高可读性. 局部空间中的变量别名只需要取原名字的最后部分.

```
/**
 * @constructor
 */
some.long.namespace.MyClass = function() {
};

/**
 * @param {some.long.namespace.MyClass} a
 */
some.long.namespace.MyClass.staticHelper = function(a) {
  ...
};

myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  var staticHelper = some.long.namespace.MyClass.staticHelper;
  staticHelper(new MyClass());
};
```

不要对命名空间创建别名.

```
myapp.main = function() {
  var namespace = some.long.namespace;
  namespace.MyClass.staticHelper(new namespace.MyClass());
};
```

除非是枚举类型, 不然不要访问别名变量的属性.

```
/** @enum {string} */
some.long.namespace.Fruit = {
  APPLE: 'a',
  BANANA: 'b'
};

myapp.main = function() {
  var Fruit = some.long.namespace.Fruit;
  switch (fruit) {
    case Fruit.APPLE:
      ...
    case Fruit.BANANA:
      ...
  }
};
```

```
myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  MyClass.staticHelper(null);
};
```

不要在全局范围内创建别名, 而仅在函数块作用域中使用.

文件名

文件名应该使用小写字符, 以避免在有些系统平台上不识别大小写的命名方式. 文件名以 `.js` 结尾, 不要包含除 `-` 和 `_` 外的标点符号(使用 `-` 优于 `_`).

自定义 toString() 方法

应该总是成功调用且不要抛异常.

可自定义 `toString()` 方法, 但确保你的实现方法满足: (1) 总是成功 (2) 没有其他负面影响.

如果不满足这两个条件, 那么可能会导致严重的问题, 比如, 如果 `toString()` 调用了包含 `assert` 的函数, `assert` 输出导致失败的对象, 这在 `toString()` 也会被调用.

延迟初始化

可以

没必要在每次声明变量时就将其初始化.

明确作用域

任何时候都需要

任何时候都要明确作用域 – 提高可移植性和清晰度. 例如, 不要依赖于作用域链中的 `window` 对象.

可能在其他应用中, 你函数中的 `window` 不是指之前的那个窗口对象.

代码格式化

展开见详细描述.

主要依照C++ 格式规范 (中文版), 针对 JavaScript, 还有下面一些附加说明.

大括号

分号会被隐式插入到代码中, 所以你务必在同一行上插入大括号. 例如:

```
if (something) {
  // ...
} else {
  // ...
}
```

数组和对象的初始化

如果初始值不是很长, 就保持写在单行上:

```
var arr = [1, 2, 3];  // No space after [ or before ].
var obj = {a: 1, b: 2, c: 3};  // No space after { or before }.
```

初始值占用多行时, 缩进2个空格.

```
// Object initializer.
var inset = {
  top: 10,
  right: 20,
  bottom: 15,
  left: 12
};

// Array initializer.
this.rows_ = [
  '"Slartibartfast" <fjordmaster@magrathea.com>',
  '"Zaphod Beeblebrox" <theprez@universe.gov>',
  '"Ford Prefect" <ford@theguide.com>',
  '"Arthur Dent" <has.no.tea@gmail.com>',
  '"Marvin the Paranoid Android" <marv@googlemail.com>',
  'the.mice@magrathea.com'
];

// Used in a method call.
goog.dom.createDom(goog.dom.TagName.DIV, {
  id: 'foo',
  className: 'some-css-class',
  style: 'display:none'
}, 'Hello, world!');
```

比较长的标识符或者数值, 不要为了让代码好看些而手工对齐.

如:

```
CORRECT_Object.prototype = {
  a: 0,
  b: 1,
  lengthyName: 2
};
```

不要这样做:

```
WRONG_Object.prototype = {
  a          : 0,
  b          : 1,
  lengthyName: 2
};
```

函数参数

尽量让函数参数在同一行上.

如果一行超过 80 字符, 每个参数独占一行, 并以4个空格缩进, 或者与括号对齐, 以提高可读性. 尽可能不要让每行超过80个字符. 比如下面这样:

```
// Four-space, wrap at 80\.  Works with very long function names, s
// renaming without reindenting, low on space.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator
  // ...
};

// Four-space, one argument per line.  Works with long function nar
// survives renaming, and emphasizes each argument.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne,
    veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
  // ...
};

// Parenthesis-aligned indentation, wrap at 80\.  Visually groups a
// low on space.
function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgur
             tableModelEventHandlerProxy, artichokeDescriptorAdapte
  // ...
}

// Parenthesis-aligned, one argument per line.  Visually groups and
// emphasizes each individual argument.
function bar(veryDescriptiveArgumentNumberOne,
             veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy,
             artichokeDescriptorAdapterIterator) {
  // ...
}
```

传递匿名函数

如果参数中有匿名函数, 函数体从调用该函数的左边开始缩进2个空格, 而不是从 function 这个关键字开始. 这让匿名函数更加易读 (不要增加很多没必要的缩进让函数体显示在屏幕的右侧).

```
var names = items.map(function(item) {
                        return item.name;
                      });

prefix.something.reallyLongFunctionName('whatever', function(a1, a2
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});
```

更多的缩进

事实上，除了初始化数组和对象，和传递匿名函数外，所有被拆开的多行文本要么选择与之前的表达式左对齐，要么以4个（而不是2个）空格作为一缩进层次。

```
someWonderfulHtml = '' +
                    getEvenMoreHtml(someReallyInterestingValues, mo
                                    evenMoreParams, 'a duck', true,
                                    slightlyMoreMonkeys(0xfff)) +
                    '';

thisIsAVeryLongVariableName =
    hereIsAnEvenLongerOtherFunctionNameThatWillNotFitOnPrevLine();

thisIsAVeryLongVariableName = 'expressionPartOne' + someMethodThatI
    thisIsAnEvenLongerOtherFunctionNameThatCannotBeIndentedMore();

someValue = this.foo(
    shortArg,
    'Some really long string arg - this is a pretty common case, ac
    shorty2,
    this.bar());

if (searchableCollection(allYourStuff).contains(theStuffYouWant) &&
    !ambientNotification.isActive() && (client.isAmbientSupported()
                                        client.alwaysTryAmbientAnyw
  ambientNotification.activate();
}
```

空行

使用空行来划分一组逻辑上相关联的代码片段。

```
doSomethingTo(x);
doSomethingElseTo(x);
andThen(x);

nowDoSomethingWith(y);

andNowWith(z);
```

二元和三元操作符

操作符始终跟随着前行，这样就不用顾虑分号的隐式插入问题．如果一行实在放不下，还是按照上述的缩进风格来换行．

```
var x = a ? b : c;  // All on one line if it will fit.

// Indentation +4 is OK.
var y = a ?
    longButSimpleOperandB : longButSimpleOperandC;

// Indenting to the line position of the first operand is also OK.
var z = a ?
        moreComplicatedB :
        moreComplicatedC;
```

括号

只在需要的时候使用

不要滥用括号, 只在必要的时候使用它.

对于一元操作符(如 `delete` , `typeof` 和 `void`), 或是在某些关键词(如 `return` , `throw` , `case` , `new`)之后, 不要使用括号.

字符串

使用 ' 优于 "

单引号 (') 优于双引号 (").

当你创建一个包含 HTML 代码的字符串时就知道它的好处了.

可见性 (私有域和保护域)

推荐使用 JSDoc 中的两个标记: `@private` 和 `@protected`

JSDoc 的两个标记 `@private` 和 `@protected` 用来指明类, 函数, 属性的可见性域.

标记为 `@private` 的全局变量和函数, 表示它们只能在当前文件中访问.

标记为 `@private` 的构造器, 表示该类只能在当前文件或是其静态/普通成员中实例化; 私有构造器的公共静态属性在当前文件的任何地方都可访问, 通过 `instanceof` 操作符也可.

永远不要为 全局变量, 函数, 构造器加 `@protected` 标记.

```
// File 1.
// AA_PrivateClass_ and AA_init_ are accessible because they are gl
// and in the same file.

/**
 * @private
 * @constructor
 */
AA_PrivateClass_ = function() {
};

/** @private */
function AA_init_() {
  return new AA_PrivateClass_();
}

AA_init_();
```

标记为 `@private` 的属性, 在当前文件中可访问它; 如果是类属性私有, "拥有"该属性的类的所有静态/普通成员也可访问, 但它们不能被不同文件中的子类访问或覆盖.

标记为 `@protected` 的属性, 在当前文件中可访问它, 如果是类属性保护, 那么"拥有"该属性的类及其子类中的所有静态/普通成员也可访问.

注意: 这与 C++, Java 中的私有和保护不同, 它们是在当前文件中, 检查是否具有访问私有/保护属性的权限, 有权限即可访问, 而不是只能在同一个类或类层次上.而 C++ 中的私有属性不能被子类覆盖.

(C++/Java 中的私有/保护是指作用域上的可访问性, 在可访问性上的限制. JS 中是在限制在作用域上. PS: 可见性是与作用域对应)

```
// File 1.

/** @constructor */
  AA_PublicClass = function() {
};

/** @private */
AA_PublicClass.staticPrivateProp_ = 1;

/** @private */
AA_PublicClass.prototype.privateProp_ = 2;

/** @protected */
AA_PublicClass.staticProtectedProp = 31;

/** @protected */
AA_PublicClass.prototype.protectedProp = 4;

// File 2.

/**
 * @return {number} The number of ducks we've arranged in a row.
 */
AA_PublicClass.prototype.method = function() {
  // Legal accesses of these two properties.
  return this.privateProp_ + AA_PublicClass.staticPrivateProp_;
};

// File 3.

/**
 * @constructor
 * @extends {AA_PublicClass}
 */
AA_SubClass = function() {
  // Legal access of a protected static property.
  AA_PublicClass.staticProtectedProp = this.method();
};
goog.inherits(AA_SubClass, AA_PublicClass);

/**
 * @return {number} The number of ducks we've arranged in a row.
 */
AA_SubClass.prototype.method = function() {
  // Legal access of a protected instance property.
  return this.protectedProp;
};
```

JavaScript 类型

强烈建议你去使用编译器.

如果使用 JSDoc, 那么尽量具体地, 准确地根据它的规则来书写类型说明. 目前支持两种JS2和 JS1.x 类型规范.

JavaScript 类型语言

JS2 提议中包含了一种描述 JavaScript 类型的规范语法, 这里我们在 JSDoc 中采用其来描述函数参数和返回值的类型.

JSDoc 的类型语言, 按照 JS2 规范, 也进行了适当改变, 但编译器仍然支持旧语法.

名称	语法	描边
普通类型	{boolean} , {Window} , {goog.ui.Menu}	普通类型的描
复杂类型	{Array.<string>} 字符串数组. {Object.<string, number>} 键为字符串, 值为整数的对象类型.	参数化类型, 该类型中包含 列"类型参数" Java 中的泛
联合类型	{(number|boolean)} 一个整数或者布尔值.	表示其值可能 型, 也可能是
记录类型	{{myNum: number, myObject}} 由现有类型组成的类型.	表示包含指定 类型的值. 这 myNum 为 类型, myObj 任意类型. 注 为类型语法的 比如, Array.<{le 表示一具有 属性的 Arr
可为空类型	{?number} 一个整型数或者为 NULL	表示一个值可 类型或者 nu 认, 每个对象 空的. 注意: 函 不可为空.
非空类型	{!Object} 一个对象, 但绝不会是 null 值.	说明一个值是 且肯定不是 n 情况下, 所有 (boolean, nu

		string, 和 un 不可为空.
函数类型	`{function(string, boolean)}` 具有两个参数 (string 和 boolean) 的函数类型, 返回值未知.	说明一个函数
函数返回类型	`{function(): number}` 函数返回一个整数.	说明函数的这
函数的 `this` 类型	`{function(this:goog.ui.Menu, string)}` 函数只带一个参数 (string), 并且在上下文 goog.ui.Menu 中执行.	说明函数类型 文类型.
可变参数	`{function(string, ...[number]): number}` 带一个参数 (字符类型) 的函数类型, 并且函数的参数个数可变, 但参数类型必须为 number.	说明函数的可 数.
可变长的参数 (使用 `@param` 标记)	`@param {...number} var_args` 函数参数个数可变.	使用标记, 说 有不定长参数
函数的 缺省参数	`{function(?string=, number=)}` 函数带一个可空且可选的字符串型参数, 一个可选整型参数. `=` 语法只针对 `function` 类型有效.	说明函数的可
函数 可选参数 (使用 `@param` 标记)	`@param {number=} opt_argument` `number` 类型的可选参数.	使用标记, 说 有可选参数.
所有类型	`{*}`	表示变量可以 类型.

JavaScript中的类型

`number`

```
1
1.0
-5
1e5
Math.PI
```

`Number`

数值对象

```
new Number(true)
```

`string`

字符串值

```
'Hello'
"World"
String(42)
```

`String`

字符串对象

```
new String('Hello')
new String(42)
```

`boolean`

布尔值

```
true
false
Boolean(0)
```

`Boolean`

布尔对象

```
new Boolean(true)
```

`RegExp`

```
new RegExp('hello')
/world/g
```

`Date`

```
new Date
new Date()
```

`null`

```
null
```

`undefined`

```
undefined
```

`void`

没有返回值

```
function f() {
  return;
}
```

`Array`

类型不明确的数组

```
['foo', 0.3, null]
[]
```

`Array.<number>`

```
[11, 22, 33]
```

`Array.<Array.<string>>`

```
Array.<Array.<string>>
```

`Object`

```
{}
{foo: 'abc', bar: 123, baz: null}
```

`Object.<string>`

```
{'foo': 'bar'}
```

`Object.<number, string>`

键为整数, 值为字符串的对象.

注意, JavaScript 中, 键总是被转换成字符串, 所以 `obj['1'] == obj[1]` . 也所以, 键在 for...in 循环中是字符串类型. 但在编译器中会明确根据键的类型来查找对象.

```
var obj = {};
obj[1] = 'bar';
```

`Function`

函数对象

```
function(x, y) {
  return x * y;
}
```

`function(number, number): number`

函数值

```
function(x, y) {
  return x * y;
}
```

`SomeClass`

```
/** @constructor */
function SomeClass() {}

new SomeClass();
```

`SomeInterface`

```
/** @interface */
function SomeInterface() {}

SomeInterface.prototype.draw = function() {};
```

`project.MyClass`

```
/** @constructor */
project.MyClass = function () {}

new project.MyClass()
```

`project.MyEnum`

枚举

```
/** @enum {string} */
project.MyEnum = {
  BLUE: '#0000dd',
  RED: '#dd0000'
};
```

`Element`

DOM 中的元素

```
document.createElement('div')
```

`Node`

DOM 中的节点.

```
document.body.firstChild
```

`HTMLInputElement`

DOM 中, 特定类型的元素.

```
htmlDocument.getElementsByTagName('input')[0]
```

可空 vs. 可选 参数和属性

JavaScript 是一种弱类型语言, 明白可选, 非空和未定义参数或属性之间的细微差别还是很重要的.

对象类型(引用类型)默认非空. 注意: 函数类型默认不能为空.

除了字符串, 整型, 布尔, undefined 和 null 外, 对象可以是任何类型.

```
/**
 * Some class, initialized with a value.
 * @param {Object} value Some value.
 * @constructor
 */
function MyClass(value) {
  /**
   * Some value.
   * @type {Object}
   * @private
   */
  this.myValue_ = value;
}
```

告诉编译器 `myValue_` 属性为一对象或 null. 如果 `myValue_` 永远都不会为 null, 就应该如下声明:

```
/**
 * Some class, initialized with a non-null value.
 * @param {!Object} value Some value.
 * @constructor
 */
function MyClass(value) {
  /**
   * Some value.
   * @type {!Object}
   * @private
   */
  this.myValue_ = value;
}
```

这样, 当编译器在代码中碰到 `MyClass` 为 null 时, 就会给出警告.

函数的可选参数可能在运行时没有定义, 所以如果他们又被赋给类属性, 需要声明成:

```
/**
 * Some class, initialized with an optional value.
 * @param {Object=} opt_value Some value (optional).
 * @constructor
 */
function MyClass(opt_value) {
  /**
   * Some value.
   * @type {Object|undefined}
   * @private
   */
  this.myValue_ = opt_value;
}
```

这告诉编译器 `myValue_` 可能是一个对象, 或 null, 或 undefined.

注意: 可选参数 `opt_value` 被声明成 `{Object=}`, 而不是 `{Object|undefined}`. 这是因为可选参数可能是 undefined. 虽然直接写 undefined 也并无害处, 但鉴于可阅读性还是写成上述的样子.

最后, 属性的非空和可选并不矛盾, 属性既可是非空, 也可是可选的. 下面的四种声明各不相同:

```
/**
 * Takes four arguments, two of which are nullable, and two of whic
 * optional.
 * @param {!Object} nonNull Mandatory (must not be undefined), must
 * @param {Object} mayBeNull Mandatory (must not be undefined), may
 * @param {!Object=} opt_nonNull Optional (may be undefined), but i
 *     must not be null!
 * @param {Object=} opt_mayBeNull Optional (may be undefined), may
 */
function strangeButTrue(nonNull, mayBeNull, opt_nonNull, opt_mayBeN
  // ...
};
```

注释

使用 JSDoc

我们使用JSDoc中的注释风格. 行内注释使用 // 变量 的形式. 另外, 我们也遵循C++代码注释风格. 这也就是说你需要:

- 版权和著作权的信息,
- 文件注释中应该写明该文件的基本信息(如, 这段代码的功能摘要, 如何使用, 与哪些东西相关), 来告诉那些不熟悉代码的读者.
- 类, 函数, 变量和必要的注释,
- 期望在哪些浏览器中执行,
- 正确的大小写, 标点和拼写.

为了避免出现句子片段, 请以合适的大/小写单词开头, 并以合适的标点符号结束这个句子.

现在假设维护这段代码的是一位初学者. 这可能正好是这样的!

目前很多编译器可从 JSDoc 中提取类型信息, 来对代码进行验证, 删除和压缩. 因此, 你很有必要去熟悉正确完整的 JSDoc .

顶层/文件注释

顶层注释用于告诉不熟悉这段代码的读者这个文件中包含哪些东西.

应该提供文件的大体内容, 它的作者, 依赖关系和兼容性信息. 如下:

```
// Copyright 2009 Google Inc. All Rights Reserved.

/**
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 * @author user@google.com (Firstname Lastname)
 */
```

类注释

每个类的定义都要附带一份注释, 描述类的功能和用法.也需要说明构造器参数.

如果该类继承自其它类, 应该使用 `@extends` 标记.

如果该类是对接口的实现, 应该使用 `@implements` 标记.

```
/**
 * Class making something fun and easy.
 * @param {string} arg1 An argument that makes this more interestin
 * @param {Array.<number>} arg2 List of numbers to be processed.
 * @constructor
 * @extends {goog.Disposable}
 */
project.MyClass = function(arg1, arg2) {
  // ...
};
goog.inherits(project.MyClass, goog.Disposable);
```

方法与函数的注释

提供参数的说明. 使用完整的句子, 并用第三人称来书写方法说明.

```
/**
 * Converts text to some completely different text.
 * @param {string} arg1 An argument that makes this more interestin
 * @return {string} Some return value.
 */
project.MyClass.prototype.someMethod = function(arg1) {
  // ...
};

/**
 * Operates on an instance of MyClass and returns something.
 * @param {project.MyClass} obj Instance of MyClass which leads to
 *     comment that needs to be wrapped to two lines.
 * @return {boolean} Whether something occured.
 */
function PR_someMethod(obj) {
  // ...
}
```

对于一些简单的, 不带参数的 getters, 说明可以忽略.

```
/**
 * @return {Element} The element for the component.
 */
goog.ui.Component.prototype.getElement = function() {
  return this.element_;
};
```

属性注释

也需要对属性进行注释.

```
/**
 * Maximum number of things per pane.
 * @type {number}
 */
project.MyClass.prototype.someProperty = 4;
```

类型转换的注释

有时, 类型检查不能很准确地推断出表达式的类型, 所以应该给它添加类型标记注释来明确之, 并且必须在表达式和类型标签外面包裹括号.

```
/** @type {number} */ (x)
(/** @type {number} */ x)
```

JSDoc 缩进

如果你在 `@param` , `@return` , `@supported` , `@this` 或 `@deprecated` 中断行, 需要像在代码中一样, 使用4个空格作为一个缩进层次.

```
/**
 * Illustrates line wrapping for long param/return descriptions.
 * @param {string} foo This is a param with a description too long
 *     one line.
 * @return {number} This returns something that has a description t
 *     fit in one line.
 */
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

不要在 `@fileoverview` 标记中进行缩进.

虽然不建议, 但也可对说明文字进行适当的排版对齐. 不过, 这样带来一些负面影响, 就是当你每次修改变量名时, 都得重新排版说明文字以保持和变量名对齐.

```
/**
 * This is NOT the preferred indentation method.
 * @param {string} foo This is a param with a description too long
 *                    one line.
 * @return {number} This returns something that has a description t
 *                  fit in one line.
 */
project.MyClass.prototype.method = function(foo) {
  return 5;
};
```

枚举

```
/**
 * Enum for tri-state values.
 * @enum {number}
 */
project.TriState = {
  TRUE: 1,
  FALSE: -1,
  MAYBE: 0
};
```

注意一下, 枚举也具有有效类型, 所以可以当成参数类型来用.

```
/**
 * Sets project state.
 * @param {project.TriState} state New project state.
 */
project.setState = function(state) {
  // ...
};
```

Typedefs

有时类型会很复杂. 比如下面的函数, 接收 Element 参数:

```
/**
 * @param {string} tagName
 * @param {(string|Element|Text|Array.<Element>|Array.<Text>)} cont
 * @return {Element}
 */
goog.createElement = function(tagName, contents) {
  ...
};
```

你可以使用 `@typedef` 标记来定义个常用的类型表达式.

```
/**
* @param {string} tagName
* @param {goog.ElementContent} contents
* @return {Element}
*/
goog.createElement = function(tagName, contents) {
...
};
```

JSDoc 标记表

`@param`

模板 & 例子：

```
@param {Type} 变量名 描述
```

如:

```
/**
* Queries a Baz for items.
* @param {number} groupNum Subgroup id to query.
* @param {string|number|null} term An itemName,
*     or itemId, or null to search everything.
*/
goog.Baz.prototype.query = function(groupNum, term) {
// ...
};
```

描述：给方法, 函数, 构造器中的参数添加说明.

类型检测支持：完全支持.

`@return`

模板 & 例子：

```
@return {Type} 描述
```

如:

```
/**
* @return {string} The hex ID of the last item.
*/
goog.Baz.prototype.getLastId = function() {
// ...
return id;
};
```

描述：给方法, 函数的返回值添加说明. 在描述布尔型参数时,

用 “Whether the component is visible” 这种描述优于 “True if the component is visible, false otherwise”.

如果函数没有返回值, 就不需要添加 `@return` 标记.

类型检测支持：完全支持.

`@author`

模板 & 例子：

```
@author username@google.com (first last)
```

如:

```
/**
* @fileoverview Utilities for handling textareas.
* @author kuth@google.com (Uthur Pendragon)
*/
```

描述：表明文件的作者,通常仅会在 `@fileoverview` 注释中使用到它.

类型检测支持：不需要.

`@see`

模板 & 例子：

```
@see Link
```

如:

```
/**
* Adds a single item, recklessly.
* @see #addSafely
* @see goog.Collect
* @see goog.RecklessAdder#add
...
```

描述：给出引用链接, 用于进一步查看函数/方法的相关细节.

类型检测支持：不需要.

`@fileoverview`

模板 & 例子：

```
@fileoverview 描述
```

如:

```
/**
* @fileoverview Utilities for doing things that require this very l
* but not indented comment.
* @author kuth@google.com (Uthur Pendragon)
*/
```

描述：文件通览.

类型检测支持：不需要.

`@constructor`

模板 & 例子：

```
@constructor
```

如:

```
/**
* A rectangle.
* @constructor
*/
function GM_Rect() {
...
}
```

描述：指明类中的构造器.

类型检测支持：会检查. 如果省略了, 编译器将禁止实例化.

`@interface`

模板 & 例子：

```
@interface
```

如:

```
/**
* A shape.
* @interface
*/
function Shape() {};
Shape.prototype.draw = function() {};

/**
* A polygon.
* @interface
* @extends {Shape}
*/
function Polygon() {};
Polygon.prototype.getSides = function() {};
```

描述：指明这个函数是一个接口.

类型检测支持：会检查. 如果实例化一个接口, 编译器会警告.

`@type`

模板 & 例子：

```
@type Type
@type {Type}
```

如:

```
/**
* The message hex ID.
* @type {string}
*/
var hexId = hexId;
```

描述：标识变量, 属性或表达式的类型.

大多数类型是不需要加大括号的, 但为了保持一致, 建议统一加大括号.|

类型检测支持：会检查

`@extends`

模板 & 例子：

```
@extends Type
@extends {Type}
```

如:

```
/**
* Immutable empty node list.
* @constructor
* @extends goog.ds.BasicNodeList
*/
goog.ds.EmptyNodeList = function() {
...
};
```

描述：与 @constructor 一起使用, 用来表明该类是扩展自其它类的. 类型外的大括号可写可不写.

类型检测支持：会检查

`@implements`

模板 & 例子：

```
@implements Type
@implements {Type}
```

如:

```
/**
* A shape.
* @interface
*/
function Shape() {};
Shape.prototype.draw = function() {};

/**
* @constructor
* @implements {Shape}
*/
function Square() {};
Square.prototype.draw = function() {
...
};
```

描述：与 @constructor 一起使用, 用来表明该类实现自一个接口. 类型外的大括号可写可不写.

类型检测支持：会检查. 如果接口不完整, 编译器会警告.

`@lends`

模板 & 例子：

```
@lends objectName
@lends {objectName}
```

如:

```
goog.object.extend(
Button.prototype,
/** @lends {Button.prototype} */ {
isButton: function() { return true; }
});
```

描述：表示把对象的键看成是其他对象的属性. 该标记只能出现在对象语法中.

注意, 括号中的名称和其他标记中的类型名称不一样, 它是一个对象名, 以"借过来"的属性名命名.

如, @type {Foo} 表示 “Foo 的一个实例”, but @lends {Foo} 表示 “Foo 构造器”.

更多有关此标记的内容见JSDoc Toolkit docs.

类型检测支持：会检查

`@private`

模板 & 例子：

```
@private
```

如:

```
/**
* Handlers that are listening to this logger.
* @type Array.<Function>
* @private
*/
this.handlers_ = [];
```

描述：指明那些以下划线结尾的方法和属性是

私有的.

不推荐使用后缀下划线, 而应改用 @private.

类型检测支持：需要指定标志来开启.

`@protected`

模板 & 例子：

```
@protected
```

如:

```
/**
* Sets the component's root element to the given element.  Conside
* protected and final.
* @param {Element} element Root element for the component.
* @protected
*/
goog.ui.Component.prototype.setElementInternal = function(element)
// ...
};
```

描述：指明接下来的方法和属性是被保护的.

被保护的方法和属性的命名不需要以下划线结尾, 和普通变量名没区别.

类型检测支持：需要指定标志来开启.

`@this`

模板 & 例子：

```
@this Type
@this {Type}
```

如:

```
pinto.chat.RosterWidget.extern('getRosterElement',
/**
* Returns the roster widget element.
* @this pinto.chat.RosterWidget
* @return {Element}
*/
function() {
return this.getWrappedComponent_().getElement();
});
```

描述：指明调用这个方法时, 需要在哪个上下文中. 当 this 指向的不是原型方法的函数时必须使用这个标记.

类型检测支持：会检查

`@supported`

模板 & 例子：

```
@supported 描述
```

如:

```
/**
* @fileoverview Event Manager
* Provides an abstracted interface to the
* browsers' event systems.
* @supported So far tested in IE6 and FF1.5
*/
```

描述：在文件概述中用到, 表明支持哪些浏览器.

类型检测支持：不需要.

@enum

模板 & 例子：

```
@enum {Type}
```

如:

```
/**
* Enum for tri-state values.
* @enum {number}
*/
project.TriState = {
TRUE: 1,
FALSE: -1,
MAYBE: 0
};
```

描述：用于枚举类型.

类型检测支持：完全支持. 如果省略, 会认为是整型.

@deprecated

模板 & 例子：

```
@deprecated 描述
```

如:

```
/**
* Determines whether a node is a field.
* @return {boolean} True if the contents of
*     the element are editable, but the element
*     itself is not.
* @deprecated Use isField().
*/
BN_EditUtil.isTopEditableField = function(node) {
// ...
};
```

描述：告诉其他开发人员, 此方法, 函数已经过时, 不要再使用. 同时也会给出替代方法或函数.

类型检测支持：不需要

`@override`

模板 & 例子：

```
@override
```

如:

```
/**
* @return {string} Human-readable representation of project.SubClas
* @override
*/
project.SubClass.prototype.toString() {
// ...
};
```

描述：指明子类的方法和属性是故意隐藏了父类的方法和属性. 如果子类的方法和属性没有自己的文档, 就会继承父类的.

类型检测支持：会检查

`@inheritDoc`

模板 & 例子：

```
@inheritDoc
```

如:

```
/** @inheritDoc */
project.SubClass.prototype.toString() {
// ...
};
```

描述：指明子类的方法和属性是故意隐藏了父类的方法和属性, 它们具有相同的文档. 注意: 使用@inheritDoc 意味着也同时使用了 @override. |

类型检测支持：会检查

`@code`

模板 & 例子：

```
{@code …}
```

如:

```
/**
* Moves to the next position in the selection.
* Throws {@code goog.iter.StopIteration} when it
* passes the end of the range.
* @return {Node} The node at the next position.
*/
goog.dom.RangeIterator.prototype.next = function() {
// ...
};
```

描述：说明这是一段代码, 让它能在生成的文档中正确的格式化.

类型检测支持：不适用.

`@license` or `@preserve`

模板 & 例子：

```
@license
```

描述：

如:

```
/**
* @preserve Copyright 2009 SomeThirdParty.
* Here is the full license text and copyright
* notice for this file. Note that the notice can span several
* lines and is only terminated by the closing star and slash:
*/
```

描述：所有被标记为 @license 或 @preserve 的, 会被编译器保留不做任何修改而直接输出到最终文挡中.

这个标记让一些重要的信息(如法律许可或版权信息)原样保留, 同样, 文本中的换行也会被保留. |

类型检测支持：不需要.

`@noalias`

模板 & 例子：

```
@noalias
```

如:

```
/** @noalias */
function Range() {}
```

描述：在外部文件中使用，告诉编译器不要为这个变量或函数重命名.

类型检测支持：不需要.

`@define`

模板 & 例子：

```
@define {Type} 描述
```

如:

```
/** @define {boolean} */
var TR_FLAGS_ENABLE_DEBUG = true;

/** @define {boolean} */
goog.userAgent.ASSUME_IE = false;
```

描述：表示该变量可在编译时被编译器重新赋值.

在上面例子中，BUILD 文件中指定了

```
–define='goog.userAgent.ASSUME_IE=true'
```

这个编译之后，常量 goog.userAgent.ASSUME_IE 将被全部直接替换为 true.

类型检测支持：不需要.

`@export`

模板 & 例子：

```
@export
```

如:

```
/** @export */
foo.MyPublicClass.prototype.myPublicMethod = function() {
// ...
};
```

描述：

上面的例子代码，当编译器运行时指定 –generate_exports 标志，会生成下面的代码:

```
goog.exportSymbol('foo.MyPublicClass.prototype.myPublicMethod',
foo.MyPublicClass.prototype.myPublicMethod);
```

编译后，将源代码中的名字原样导出.

使用 @export 标记时，应该

1. 包含 //javascript/closure/base.js，或者
2. 在代码库中自定义 goog.exportSymbol 和 goog.exportProperty 两个方法，并保证有相同的调用方式.

类型检测支持：不需要.

@const

模板 & 例子：

```
@const
```

如:

```
/** @const */ var MY_BEER = 'stout';

/**
* My namespace's favorite kind of beer.
* @const
* @type {string}
*/
mynamespace.MY_BEER = 'stout';

/** @const */ MyClass.MY_BEER = 'stout';
```

描述：

声明变量为只读，直接写在一行上.

如果其他代码中重写该变量值, 编译器会警告.

常量应全部用大写字符, 不过使用这个标记, 可以帮你消除命名上依赖.

虽然 jsdoc.org 上列出的 @final 标记作用等价于 @const , 但不建议使用.

@const 与 JS1.5 中的 const 关键字一致.

注意, 编译器不禁止修改常量对象的属性(这与 C++ 中的常量定义不一样).

如果可以准确推测出常量类型的话，那么类型申明可以忽略. 如果指定了类型, 应该也写在同一行上.

变量的额外注释可写可不写.

类型检测支持：支持.

`@nosideeffects`

模板 & 例子：

```
@nosideeffects
```

如:

```
/** @nosideeffects */
function noSideEffectsFn1() {
// ...
};

/** @nosideeffects */
var noSideEffectsFn2 = function() {
// ...
};

/** @nosideeffects */
a.prototype.noSideEffectsFn3 = function() {
// ...
};
```

描述：用于对函数或构造器声明, 说明调用此函数不会有副作用. 编译器遇到此标记时, 如果调用函数的返回值没有其他地方使用到, 则会将这个函数整个删除.

类型检测支持：不需要检查.

`@typedef`

模板 & 例子：

```
@typedef
```

如:

```
/** @typedef {(string|number)} */
goog.NumberLike;

/** @param {goog.NumberLike} x A number or a string. */
goog.readNumber = function(x) {
...
}
```

描述：这个标记用于给一个复杂的类型取一个别名.

类型检测支持：会检查

`@externs`

模板 & 例子：

```
@externs
```

如:

```
/**
* @fileoverview This is an externs file.
* @externs
*/

var document;
```

描述：

指明一个外部文件.

类型检测支持：不会检查

在第三方代码中，你还会见到其他一些 JSDoc 标记. 这些标记在JSDoc Toolkit Tag Reference都有介绍到，但在 Google 的代码中，目前不推荐使用. 你可以认为这些是将来会用到的 "保留" 名. 它们包含:

- @augments
- @argument
- @borrows
- @class
- @constant

- @constructs
- @default
- @event
- @example
- @field
- @function
- @ignore
- @inner
- @link
- @memberOf
- @name
- @namespace
- @property
- @public
- @requires
- @returns
- @since
- @static
- @version

JSDoc 中的 HTML

类似于 JavaDoc, JSDoc 支持许多 HTML 标签, 如 `<code>` , `<pre>` , `<strong>` , `<ul>` , `<ol>` , `<li>` , `<a>` , 等等.

这就是说 JSDoc 不会完全依照纯文本中书写的格式. 所以, 不要在 JSDoc 中, 使用空白字符来做格式化:

```
/**
 * Computes weight based on three factors:
 *   items sent
 *   items received
 *   last timestamp
 */
```

上面的注释, 出来的结果是:

```
Computes weight based on three factors: items sent items received i
```

应该这样写:

```
/**
 * Computes weight based on three factors:
 * <ul>
 * <li>items sent
 * <li>items received
 * <li>last timestamp
 * </ul>
 */
```

另外, 也不要包含任何 HTML 或类 HTML 标签, 除非你就想让它们解析成 HTML 标签.

```
/**
 * Changes <b> tags to  tags.
 */
```

出来的结果是:

```
Changes tags to tags.
```

另外, 也应该在源代码文件中让其他人更可读, 所以不要过于使用 HTML 标签:

```
/**
 * Changes &lt;b&gt; tags to &lt;span&gt; tags.
 */
```

上面的代码中, 其他人就很难知道你想干嘛, 直接改成下面的样子就清楚多了:

```
/**
* Changes 'b' tags to 'span' tags.
*/
```

编译

推荐使用

建议您去使用 JS 编译器, 如 Closure Compiler.

Tips and Tricks

JavaScript 小技巧

True 和 False 布尔表达式

下面的布尔表达式都返回 false:

- `null`
- `undefined`
- `''` 空字符串
- `0` 数字0

但小心下面的, 可都返回 true:

- `'0'` 字符串0
- `[]` 空数组
- `{}` 空对象

下面段比较糟糕的代码:

```
while (x != null) {
```

你可以直接写成下面的形式(只要你希望 x 不是 0 和空字符串, 和 false):

如果你想检查字符串是否为 null 或空:

```
if (y != null && y != '') {
```

但这样会更好:

注意: 还有很多需要注意的地方, 如:

- `Boolean('0') == true` `'0' != true`
- `0 != null` `0 == []` `0 == false`
- `Boolean(null) == false` `null != true` `null != false`
- `Boolean(undefined) == false` `undefined != true` `undefined != false`
- `Boolean([]) == true` `[] != true` `[] == false`
- `Boolean({}) == true` `{} != true` `{} != false`

条件(三元)操作符 (?:)

三元操作符用于替代下面的代码:

```
if (val != 0) {
  return foo();
} else {
  return bar();
}
```

你可以写成:

在生成 HTML 代码时也是很有用的:

```
var html = '<input type="checkbox"' +
    (isChecked ? ' checked' : '') +
    (isEnabled ? '' : ' disabled') +
    ' name="foo">';
```

`&&` 和 `||`

二元布尔操作符是可短路的, 只有在必要时才会计算到最后一项.

“||” 被称作为 ‘default’ 操作符, 因为可以这样:

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win;
  if (opt_win) {
    win = opt_win;
  } else {
    win = window;
  }
  // ...
}
```

你可以使用它来简化上面的代码:

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

“&&” 也可简短代码.比如:

```
if (node) {
  if (node.kids) {
    if (node.kids[index]) {
      foo(node.kids[index]);
    }
  }
}
```

你可以像这样来使用:

```
if (node && node.kids && node.kids[index]) {
  foo(node.kids[index]);
}
```

或者:

```
var kid = node && node.kids && node.kids[index];
if (kid) {
  foo(kid);
}
```

不过这样就有点儿过头了:

```
node && node.kids && node.kids[index] && foo(node.kids[index]);
```

使用 join() 来创建字符串

通常是这样使用的:

```
function listHtml(items) {
  var html = '';
  for (var i = 0; i < items.length; ++i) {
    if (i > 0) {
      html += ', ';
    }
    html += itemHtml(items[i]);
  }
  html += '';
  return html;
}
```

但这样在 IE 下非常慢, 可以用下面的方式:

```
function listHtml(items) {
  var html = [];
  for (var i = 0; i < items.length; ++i) {
    html[i] = itemHtml(items[i]);
  }
  return '' + html.join(', ') + '';
}
```

你也可以是用数组作为字符串构造器, 然后通过 `myArray.join('')` 转换成字符串. 不过由于赋值操作快于数组的 `push()` , 所以尽量使用赋值操作.

遍历 Node List

Node lists 是通过给节点迭代器加一个过滤器来实现的.

这表示获取他的属性, 如 length 的时间复杂度为 O(n), 通过 length 来遍历整个列表需要 O(n^2).

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0; i < paragraphs.length; i++) {
  doSomething(paragraphs[i]);
}
```

这样做会更好:

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {
  doSomething(paragraph);
}
```

这种方法对所有的 collections 和数组(只要数组不包含 falsy 值) 都适用.

在上面的例子中, 也可以通过 firstChild 和 nextSibling 来遍历孩子节点.

```
var parentNode = document.getElementById('foo');
for (var child = parentNode.firstChild; child; child = child.nextS
  doSomething(child);
}
```

Parting Words

保持一致性.

当你在编辑代码之前, 先花一些时间查看一下现有代码的风格. 如果他们给算术运算符添加了空格, 你也应该添加.

如果他们的注释使用一个个星号盒子, 那么也请你使用这种方式.

代码风格中一个关键点是整理一份常用词汇表, 开发者认同它并且遵循, 这样在代码中就能统一表述.

我们在这提出了一些全局上的风格规则, 但也要考虑自身情况形成自己的代码风格.

但如果你添加的代码和现有的代码有很大的区别, 这就让阅读者感到很不和谐.

所以, 避免这种情况的发生.